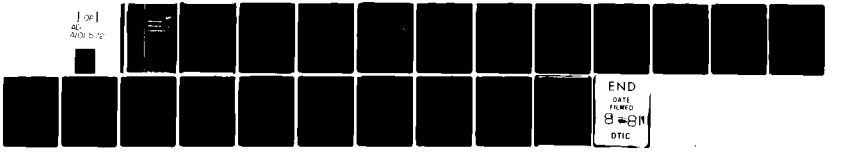
AD-A101 572 NATIONAL PHYSICAL LAB TEDDINGTON (ENGLAND) DIV OF C--ETC F/G 9/2
MODIFICATIONS TO THE PL-156 COMPILER DURING 1971,(U)
JAN 72 D A BELL
UNCLASSIFIED NPL-COM SCI-54 NL



National Physical Laboratory

Division of Computer Science

MODIFICATIONS TO THE PL-516 COMPILER DURING 1971

by Donald A. Bell

Department of Trade and Industry

SP8/61

Approved on behalf of Director, NPL by
Mr. D.W. Davies, Superintendent, Division of Computer Science

(11) Jan 73 (14) NPL-COM SCI-54

(6) Modifications to the PL-516 Compiler during 1971,

(10) Donald A Bell

(12) 11

Summary

This report gives details of the changes in syntax of PL-516 and the additional constructions added since the publication of Com. Sci. Report No 44 and CCU Report No 9.

The principal changes are the use of := for "becomes"; an explicit block structure around several declarations; case statements and expressions; more elaborate for clauses; simple conditional labels; a range test in conditions, the binary operators min and max, and a few other minor changes.

With the exception of the := change, programs which compiled successfully with the older system will still work.

NWC 412441

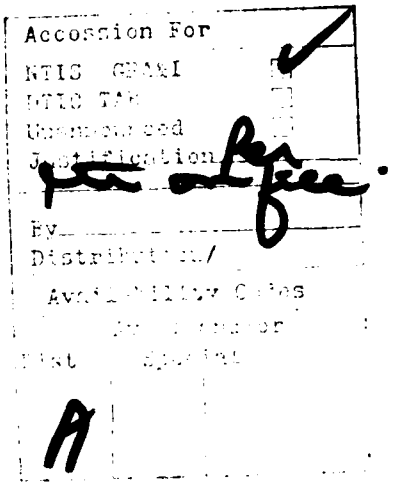
Contents

1. Assignment Statements

The := symbol is now used for assignments, the = must follow the : without any intervening space or other character. Constants, switches etc still use the = symbol when setting up values and conditions retain it as before. An example will make this clear

```
integer       I, J, K;
compconst     C = 3, F = 4;
constant      Z = '260;
switch        SS = L1, L2, L3;
string        S = 'MESSAGE';
begin

              I := C,
L3:           J, K := F + 7;
              when I = F then goto SS[K];
L1: L2:
end;
```



This follows more closely the ALGOL 60 usage, except for switches.

2. Case Statements and Expressions

Where one of a series of statements is to be executed according to the setting of a parameter, then the use of a case statement eliminates the need for switches and labels. The syntax is as follows

<case statement> ::= case <cell> of (<case body>)

<case body> ::= < <statement> list <,> >

That is the statements are separated by commas (not semicolons) and all enclosed in round brackets

For example if the statements to be executed are:

```
I := 3
J := 7 * X + J
K := if B then Y else Z + Z2
```

according as the integer C is -3, -2 or -1 then the construction is

case C of (I := 3, J := 7 * X + J,

K := if B then Y else Z + Z2)

A statement may of course be an assignment, procedure call, compound statement, for statement or even another case statement. This last possibility is the chief reason for the round brackets.

The code generated for

case C of (S1, S2, S3)

is as follows:

```
            LDX     C
            JMP*    L,1
L1          <S1>
            JMP     L
L2          <S2>
            JMP     L
L3          <S3>
            JMP     L
            DAC     L1
            DAC     L2
            DAC     L3
L           ...
```

The implicit switch is put at the end of the series of statements and called into action by the indirect indexed jump JMP* L,1

A similar construction is applied for case expressions

<case expression> ::= case <cell> of (<case expression body>)

<case expression body> ::= < <expression> list <,> >

An example is as follows

I := case C of (1, J + K, A [XX] -Q, P(Z), -4);

The integer C may take on the values -5 to -1 to select any one

of the 5 listed expressions.

An expression may be arbitrarily complicated and may even be another case expression. The code generated for:

case C of (E1, E2, E3)

where the E's are expressions is exactly like that generated for the case statement above, with <E1> substituted for <S1> etc.

3. For Statements

The simple for statement

for <variable> := <expression> do <statement>

is inadequate to cope with the cases where

a) the final value is not -1

b) the step is not +1

Three new forms are therefore introduced, they are

I) for <variable> := <expression> to <cell> do <statement>

II) for <variable> := <expression> step <expression> until <cell> do <statement>

III) for <variable> := <expression> stepdown <expression> until <cell> do <statement>

In the first case the IRS instruction is used to step on the controlled variable, but a check is applied to see if it has passed the final value before the statement is executed. Because of the possibility of the IRS instruction producing a skip if the value changes from -1 to 0, care should be taken to see that the initial and final values of the controlled variable are of the same sign. In particular a final value of 0 should never be used. If the skip does occur, the program will escape from the for loop prematurely.

The code generated for a sample for loop is as follows:

```
for  A := B + C to D do S;
        LDA     B
        ADD     C
        STA     A
L2      LDA     A
        CAS     D
        JMP     L
        NOP
        <S>
        IRS     A
        JMP     L2
L       ...
```

The IRS instruction is put at the end of the loop so that a skip will have a definable effect (ie an escape). It should be noted that unlike the simple form of the for loop, this one may be executed zero times if the final value is less than the starting value.

The second type of for loop permits an expression for the increment, and since the controlled variable is changed by an ADD instruction rather than an IRS, its value may pass from negative to positive without trouble. The only proviso is that the value of the controlled variable must be less than or equal to the final value for the loop to proceed. Normally this will imply that the step is positive. This differs from the ALGOL 60 case where the step may be positive or negative and the test to be applied depends upon this sign. To cater for the negative case the step must be specified as stepdown. This merely controls the test to be applied, it is the programmer's responsibility to make sure that the step is really negative.

ie for I := 10 stepdown 1 until 1 do S

is a logical error, it should read

```
for  I := 10 stepdown -1 until 1 do S
```

If the until symbol is missing the error number is 160.

The code generated by examples is as follows.

Example 1.

```
for A := B + C step D - E until F do S
            LDA     B
            ADD     C
            JMP     L2
       L3   LDA     D
            SUB     E
            ADD     A
       L2   STA     A
            CAS     F
            JMP     L
            NOP
            <S>
            JMP     L3
       L    ...
```

Example 2.

for A := B + C stepdown D - E until F do S; The first 8 instructions down to the CAS are as before, continuing from L2 we have:

```
            ...
       L2   STA     A
            CAS     F
            NOP
            SKP
            JMP     L
            <S>
            JMP     L3
```

4. Simple conditional labels

Consider the statement:

if I z then goto L else goto M;

The code generated is

```
        LDA   I
        SZE
        JMP   L2
        JMP   L
        JMP   L3
L2      JMP   M
L3
```

Clearly the jump to L3 will never be executed so the program can be improved to:

when I z then goto L;
goto M;

which removes the JMP L3 instruction. However the JMP L2 could equally well have been replaced by the JMP M instruction, saving a further word. This new construction is written

goto if <condition> then <label> else <label>

The first label may be a simple label or a switch element, without affecting the form of code.

The second may also be a label or switch element, but if it is a switch element (even with subscript [#]) the extra jump will be inserted. A few examples will clarify this

goto if I z then L else M

```
        LDA   I
        SZE
        JMP   M
        JMP   L
```

```
goto if I z then SS[J] else M
        LDA     I
        SZE
        JMP     M
        LDX     J
        JMP*    SS
goto if I z then L else SS[K]
        LDA     I
        SZE
        JMP     L2
        JMP     L
    L2  LDX     K
        JMP*    SS
```

5. Efficient Conditional Expressions

Consider the expression in the statement

J := if I lz then K else @;

The code generated is:

```
        LDA     I
        SMI
        JMP     L2
        LDA     K
        JMP     L2
    L2  ...
```

The redundant JMP L2 is a jump around the code for "@" which of course does not exist. Instead of writing "else @" at the end of such a conditional expression one may write elseacc which is logically equivalent but removes the unnecessary jump.

eg J := if I lz then K elseacc;

which gives

```
        LDA     I
        SMI
        JMP     L2
        LDA     K
L2      ...
```

6. Revised procedure format

The syntax of procedure has been slightly altered from

<procedure>::= procedure <identifier>;
{<declarations>} <compound statement>

to

::= procedure <identifier>;
{<declarations>} <statement>

ie if the procedure body is a single statement, the begin and end brackets may be left out. They must still appear round the main program.

A common application of this is:

```
procedure  newline;
constant   CRLF = '106612;
    out 2 (CRLF);
```

7. Explicit Block Structure

Hitherto, if two procedures required access to the same piece of working store, that store had to be globally declared. Likewise nested procedures were expressly forbidden. A more elaborate block structure has been introduced to take the pressure off sector 0. It will be especially useful for system type programs which have to co-exist with other programs. A block of declarations may be enclosed in the brackets block and endblock eg

```
block
    integer    I, J, K;
    procedure  P;
    begin
        I := K;
        K := J;
```

```
        end;
        array      A[3];
        procedure  Q;
        begin
            K := 3;
            I := A[-3];
        end;
    endblock;
```

Unless any special action is taken, all of the identifiers will be local to the block, ie inaccessible from outside it. Clearly at least one procedure must be accessible or the whole block will be redundant. The procedure names may be declared at global level by a forward declaration even if the procedures themselves are apparently local to a block.

eg

```
        forward procedure  P, Q;
        block
            integer    I, J, K;
            procedure  P;
                I := not J;
            procedure  Q;
                J := not K;
            procedure  R;
                K := not I;
        endblock;
```

The procedures P and Q will each have an access word in sector 0, whereas procedure R will have its access word in the current sector. The identifiers I, J, K and R will be inaccessible from outside the block.

This use of the forward declaration facility to get at a procedure within a block means that it is forbidden to have a global procedure and a

local procedure with the same name. (ALGOL 60 permits this).

A procedure may now have procedures of its own which are local to it, they are declared with the other declarations eg

```
procedure  GLOBAL;
   integer  I;
      procedure  LOCAL 2;
           I := J;
      procedure  LOCAL 3;
           I := neg J;
begin        LOCAL 2;
             LOCAL 3;
end;
```

Although it was formerly forbidden to use origin declarations at local level, this is now permitted. It is worth pointing out that the facility should be used with care and only to a location in the same sector, preferably it should not be used among the declarations of a procedure since it will cause automatic dumping of the constants pool.

A local array may be declared to be later if it is necessary to separate the array word from the body of the array. An error will occur if the end of the block is reached without the set declaration, but other somewhat risky constructions are permitted by the compiler since it is assumed the facilities will only be used in carefully thought out situations.

8. A range test for numbers

A common test for numbers is to see if they lie within a certain range. The test is usually written something like this:

```
if N ge A and @ le B then .....
```

where the code begins like this

```
        LDA     N
        CAS     A
        NOP
        SKP
        JMP     L1
        CAS     B
L1      JMP     L
        NOP
        ...
```

and L is the beginning of the else clause. Machine code programmers can do much better than this by writing:

```
        LDA     N
        CAS     A
        NOP
        CAS     B
        JMP     L
        NOP
        ...
```

Since the construction is fairly common it is now in PL-516 as follows

<range condition> ::= <expression> range <cell> to <cell>

where the first <cell> has the smaller value.

There is one petty restriction however. The first <cell> can be anything permitted by the normal <cell> syntax, integer, constant, array element etc but the second <cell> must generate only one word of code (to fit the double CAS construction). This effectively excludes an array or table element unless it has a subscript of the form [#] or [#, #]. The compiler checks this length of code and will give a failure 463 if it

is exceeded. It was thought that it was better to have the facility with the restriction than not at all.

9. The max and min binary operators

To load A with the maximum value of B and C the usual construction is

```
A := if B > C then B else C;
```

This generates a total of 8 words of code, or 7 if the second B is replaced by @. In machine code it would be written as:

```
LDA     B
CAS     C
NOP
SKP
LDA     CC
```

or only 5 words of code. The code for calculating the minimum value is one word shorter in each case.

By introducing two new binary operators min and max, the efficient code is made available in PL-516. Although it may seem a little strange at first the two operators (which are both symmetric) may be used just like + or *.

eg `A := B max C`

will generate machine code like the above and

`A := B min C` will give

```
LDA     B
CAS     C
LDA     C
NOP
STA     A
```

The addition of these two brings the count of binary operators to 9.

<binary operator> ::= +|*|and|nev|max|min|-|/|mod

of which the first six are symmetric.

10. Explicit strings

Since strings are usually handled by their codewords and are often called upon at only one point in a program, it is now possible to specify them directly without using a string identifier. The definition of <term> has been extended to permit a string enclosed between double quote (") characters. When the code is executed the accumulator is loaded with the codeword of the string and a jump is taken round the string. eg the procedure call

```
OUTSTRING ("MESSAGE")
```

generates the code (in DAP)

```
        LDA     *+2
        JMP     L
        DAC     *+1,1
        BCI     4, MESSAGE"
L       JST*    OUTSTRING
```

This is one word more than would be generated by a normal string declaration, because of the jump. However the increased clarity of the program compensates for this. Where there is an even number of characters in the string, they will be packed two to a word, without the quote symbols, but if the number of characters is odd the terminating quote is included as a filler.

11. Exchange Statements

A fairly common sequence of instructions for exchanging the contents of two store locations A and B is something like this:

```
TEMP := A;
A := B;
B := TEMP;
```

generating 6 words of code and using one temporary store location. Since

an exchange instruction (IMA) exists on the DDP-516, it would be better to use this, giving (in DAP)

```
LDA    A
IMA    B
STA    A
```

An exchange statement construction has therefore been introduced as follows

<exchange statement> ::= <lhs> :=: <cell>

where <lhs> may be any of the allowed forms of the left hand side of an assignment statement: integer, array, table, @ or indirect integer or constant. The :=: symbol which reads "is exchanged with" is written without any spaces between the colons and equals symbols.

There may however only be one item to the left of the :=: symbol and the insertion of several items separated by commas will give error number 464.

ie A, B :=: C

is allowed (multiple assignments), but

A, B :=: C

is not allowed. The error will however only be detected after the :=: has been read.

11.1 The Code Produced

Properly used, the exchange operator :=: will give very efficient code but it is possible to use it in inefficient ways. For a straightforward swop of two integers A and B, the statement:

A :=: B

gives the code:

```
LDA    A
IMA    B
STA    A
```

If the first item is the @ symbol

```
@ :=: B
```

then only one word is generated: IMA B. (Note that B :=: @ is an error).

Where arrays are used the subscript calculation is repeated only if it is needed i.e.

```
Y[J] :=: B
```

generates:

```
LDX   J
LDA*  Y
IMA   B
STA*  Y
```

There is no LDX between the IMA B and the STA* Y since it is not needed, but if the text had read:

```
Y[J] :=: Z[K]
```

then the code would have been:

```
LDX   J
LDA*  Y
LDX   K
IMA*  Z
LDX   J
STA*  Y
```

with LDX J occurring twice.

The rule therefore is to keep the simplest construction on the left of the :=: symbol e.g.

```
T[K,J] :=: Y[A]
```

generates 12 words of code, whereas:

$$Y[A] :=: T[K,J]$$

achieving the same effect, generates only 9 words.

If subscripts appear on only one side of an exchange then it does not matter which way round the statement is written i.e.

$$T[K,J] :=: B \quad \text{and} \quad B :=: T[K,J]$$

both generate 7 words of code.

12. Miscellaneous

Constants set equal to a single ASC II character in the lower half of the word which had to be explicitly written out in octal can now be introduced by the double dollar $$. eg instead of '303, $$C.

An assignment statement which is to the B-register cannot also assign to a variable in core. This removes an undetected source of error in earlier versions of the compiler.

Where the syntax allowed indirect addressing through an integer, it is now possible to use a constant the same way.

The input routine has been modified to read complete lines at a time rather than single characters as formerly. The character ! (shift and 1) may be used as a backspace character eg

PRUCE!!OCEDURE will be read as PROCEDURE

When the - > typewriter facility is in use it is necessary to type a carriage return symbol before the computer responds. If a failure occurs, the trace routine will print only the 20 characters processed before the failure. Generally this will not correspond to all the characters actually read.

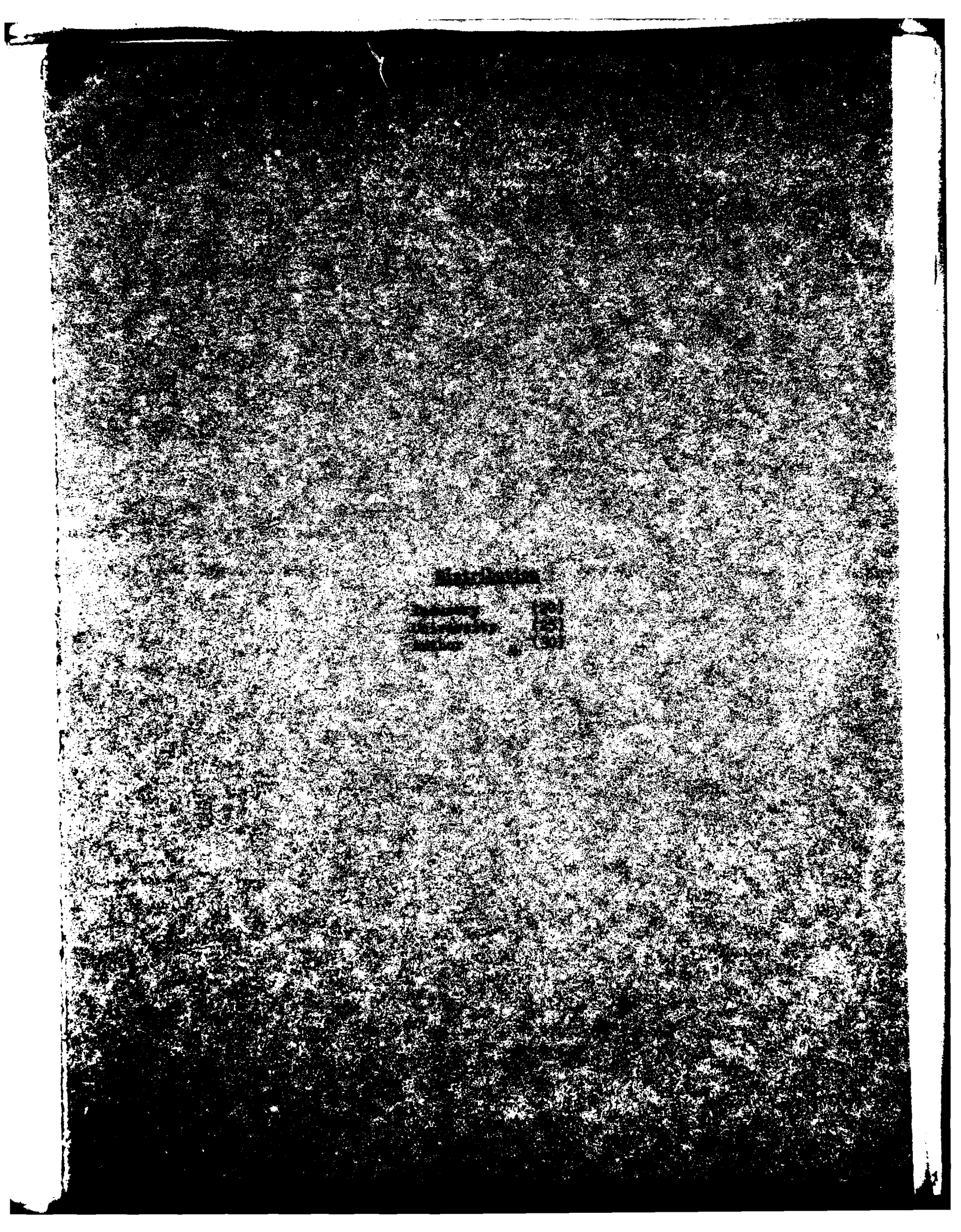
The use of sense key 4 has been changed. The global and origin declarations now give a print out anyway and key 4 if set will cause the

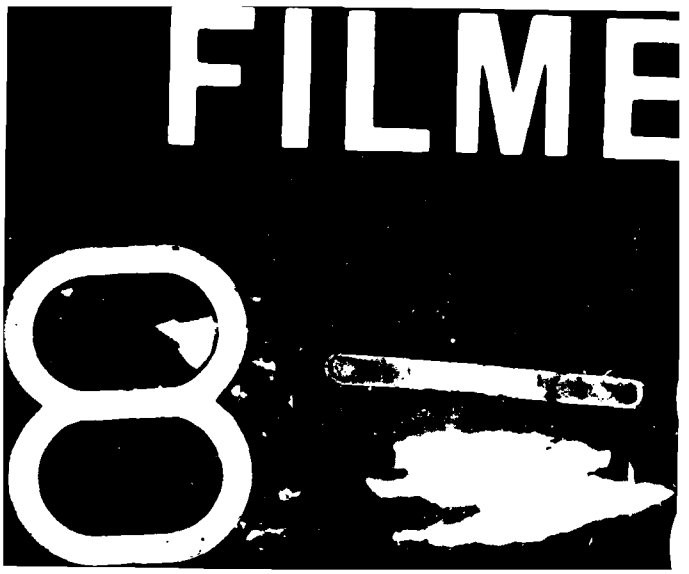
local address to be printed out on the declaration of each procedure. This will be either the address of the first word of store local to the procedure or its starting address if there is no local store.

13. References

B.A. Wichmann "PL-516, An ALGOL - like Assembly Language for the DDP-516" NPL Report C.C.U. 9 (1970)

D.A. Bell "Collected Papers on the Development of the PL-516 Programming Language" NPL Report Com. Sci. 44 (1971).




FILME
8